

JS Advance

Assignment by reference

Please take out your paper again and do a T-diagram exercise, on your paper, to predict the output for the following code. It's important that you go through this slowly one by one.

Exercise 1

```
let a = 5;
let b = a;
a = 100;
console.log('a is now', a);
console.log('b is now', b);
```

Exercise 2

```
let a = [5];
let b = a;
a.push(100);
console.log('a is now', a);
console.log('b is now', b);
```

Are you surprised both a and b are now [5,100]? Isn't that weird? The only thing that was changed was the value in a. How did b also get changed?

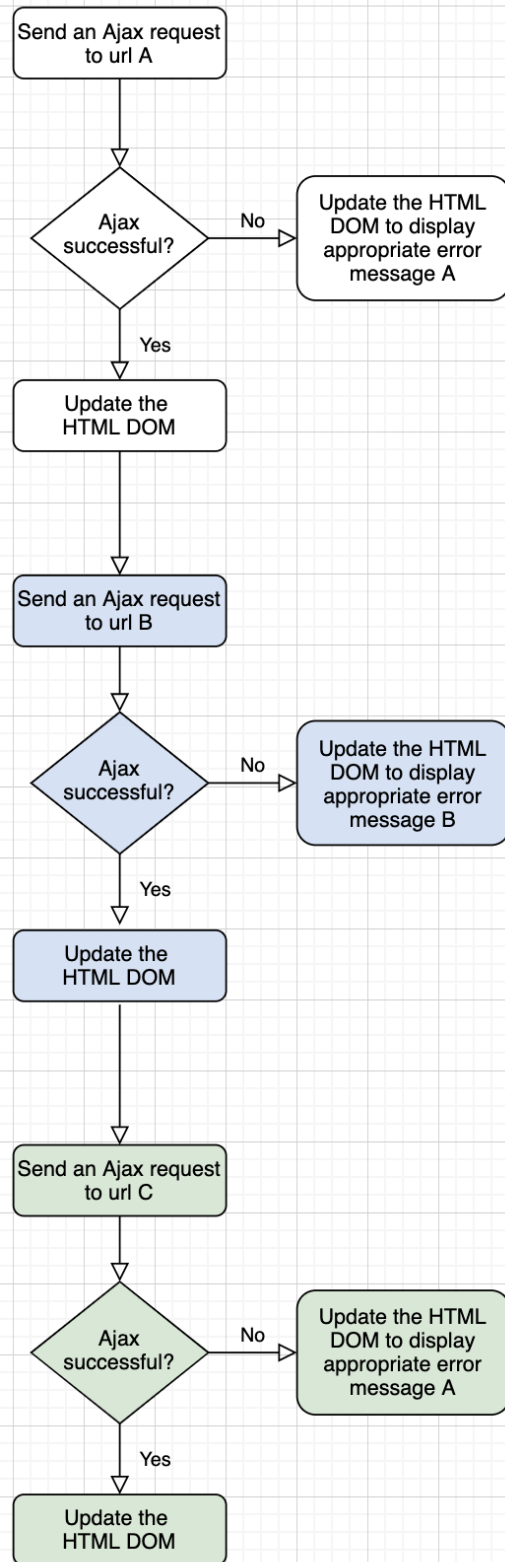
This is an important concept to understand and it's called an assignment by reference.

More about assignment by reference

If you're creating a string or a number and storing these in a variable, these get created in what's called a call stack. Whenever you store an array, an object, a function, or a class in a variable, these values get created in what's called a heap stack. Why computer program does assignment this way requires a bit more explanation on how computer was designed, which is discussed in the video.

For example,

```
let a = 5; //creates this value in the call stack
let b = a; //duplicates the value stored in a and assigns that to b
b = b + 5; //b is updated to be 10.
console.log(a, b); //logs 5 and 10
let c = "Michael"; //creates this value in the call stack
let d = c; //duplicates the value stored in c and stores that to d
d = d + " Choi"; //d is updated as "Michael Choi"
console.log(c,d); //logs "Michael" and "Michael Choi"
let x = [1,3,5]; //creates this value in the heap stack
let y = x; //y also points to that value in the heap stack
y.push(7); //at where y is pointed to, it pushes a new value called 7
console.log(x, y); //logs [1,3,5,7] twice!
```

How would you write a program to do this?

Arrow function

Async and Await

Go ahead and spend up to 15 minutes reading about async and await.

This was introduced as part of ECMAScript 2017 (and therefore is quite new). A lot of older Javascript libraries and browsers may not support async and await yet, although the latest node.js will support this feature. The browser, on the front end, may not support this feature yet, which is why it's safer to use this with Node.js but not yet on the front-end.

A good article for you to read about async and await is here:

- https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Async_await

Real World Example

For example, for Hacker Hero, when we were using callbacks, our model method looked as follows:

```
games.reorderGame = (post_data, result) => {
  connection.query(`UPDATE games_order SET game_ids_order = ? WHERE id = 1`,
    [post_data.order_list], (err, update_games_result) => {
      if(!err)
        result(err, {status : true, result : update_games_result})
      else
        result(err, {status: false, result : err});
    })
}
```

Note that we're passing a callback function as an argument to the query method.

When we re-factored the code to use async and await, it was changed to look as follows:

```
games.reorderGame = async (post_data, req=undefined) => {
  let databaseQueryModel = new DatabaseQueryModel(req);
  let updateGameOrdersQuery = mysql.format(
    `UPDATE settings SET game_ids_order = ? WHERE id = 1`,
    [post_data.game_ids_order]
  );
  try {
    let result = await databaseQueryModel.executeQuery("Games Model | updateGameOrders", u
pdateGameOrdersQuery);
    return { status : true, result : result} ;
  } catch(err) {
    return { status: false, message: "Failed to reorder games", err: err} ;
  }
}
```

Note how using async and await makes the code a bit more readable.

The code-readability improves significantly as the logic becomes more complex and as you need to do things based on series of events. For example, take a look at the code below:

```

games.fetchAdminGames = (result) => {
  connection.query(`
    SELECT games.*, heroes.hero_name, companies.name as company_name, difficulty_levels.level
as level_difficulty_name, sublevels.challenge_ids_order,
    games.level_ids_order as subtopic_ids, count(challenges.id) as challenge_ids_order
  FROM games
  LEFT JOIN difficulty_levels ON difficulty_levels.id = games.difficulty_level_id
  LEFT JOIN heroes ON heroes.id = games.hero_id
  LEFT JOIN companies ON companies.id = games.company_id
  LEFT JOIN levels ON find_in_set(levels.id, games.level_ids_order)
  LEFT JOIN sublevels ON find_in_set(sublevels.id, levels.sublevel_ids_order)
  LEFT JOIN challenges ON challenges.sublevel_id = sublevels.id AND challenges.language_id =
1 AND challenges.challenge_type_id != 3
  LEFT JOIN challenge_groups ON challenge_groups.id = challenges.challenge_group_id
  JOIN games_order ON find_in_set(games.id, games_order.game_ids_order)
  WHERE (games.is_archived IS NULL OR games.is_archived = 0)
  AND (challenge_groups.is_archived IS NULL OR challenge_groups.is_archived = 0)
  GROUP BY games.id
  ORDER BY find_in_set(games.id, games_order.game_ids_order);
`, (err, priority_order_games) => {
  if(err)
    result(err, {status : false, message : "Failed to fetch games."});
  else{
    let all_subtopics = [];
    let fetched_games = [];
    priority_order_games.map(game =>{
      if(game.subtopic_ids){
        all_subtopics.push(game.subtopic_ids);
      }

      game.total_challenges = game.challenge_ids_order;
      fetched_games.push({...game, total_subtopics : 0 });
    });
    if(all_subtopics.length > 0){
      connection.query("SELECT * FROM levels WHERE levels.is_archived IS NULL AND f
ind_in_set(levels.id, '"+ all_subtopics.join(',') +"'", (st_err, subtopic_response) => {
        if(err)
          result(err, {status : false, message : "Failed to fetch games."});
        else{
          if(subtopic_response){
            fetched_games = priority_order_games.map(game =>{
              let game_levels = (game.level_ids_order) ? game.level_ids_orde

r.split(',') : [];

              game_levels = game_levels.map(game_level =>{
                return parseInt(game_level);
              });
              let game_subtopics = subtopic_response.filter(subtopic => {
                return (game_levels.indexOf(subtopic.id) > -1);
              });
              return { ...game, total_subtopics : game_subtopics.length };
            });
            result(err, {status : true, result : fetched_games});
          }
        }
      });
    }
    else
      result(err, {status : true, result : fetched_games});
  }
})
}

```

This was re-factored and also using async/await, we got out of the callback hell behavior. This was the re-factored code afterwards.

```

// function used to fetch all the games or a specific game for the admin dashboard/challenges page
// refactored by MC on June 2020
GetAdminGames = async (game_id = undefined, req=undefined) => {
  let databaseQueryModel = new DatabaseQueryModel(req);
  let fetchAdminGamesquery = mysql.format(`
    SELECT
      games.id, games.game_type_id, games.title, games.description, games.is_beta, games
.is_visible, games.cache_players_count, games.cache_ratings_json,
      games.cache_challenges_count_json, games.thumbnail_url, games.game_url, games.introduction,
      games.language_ids_order, heroes.hero_name, companies.name as company_name,
      games.level_ids_order as subtopic_ids,
      count(levels.id) as total_subtopics, FIND_IN_SET(games.id, (SELECT game_ids_order
FROM settings)) as game_order
    FROM games
    INNER JOIN heroes ON heroes.id = games.hero_id
    INNER JOIN companies ON companies.id = games.company_id
    LEFT JOIN levels ON levels.game_id = games.id AND levels.is_archived = 0
    WHERE games.is_archived = 0
    GROUP BY games.id
    ORDER BY game_order ASC;`
  );
  if(game_id != undefined){
    fetchAdminGamesquery = mysql.format(
      `SELECT
        games.id, games.game_type_id, games.title, games.description, games.is_beta, games.is_visible,
        games.cache_players_count, games.cache_ratings_json, games.cache_challenges_count_json,
        games.thumbnail_url, games.game_url, games.introduction, games.language_ids_order,
        heroes.hero_name, companies.name as company_name,
        GROUP_CONCAT(levels.id) as level_ids, count(levels.id) as total_subtopics, games.level_ids_order
      FROM games
      LEFT JOIN levels ON levels.game_id = games.id
      INNER JOIN heroes ON heroes.id = games.hero_id
      INNER JOIN companies ON companies.id = games.company_id
      WHERE games.id = ?
      GROUP BY games.id
      `, [game_id]
    );
  }
  try {
    let game_result = await databaseQueryModel.executeQuery(`Games Model | GetAdminGames (${game_id})`, fetchAdminGamesquery);
    let fetchLanguageQuery = mysql.format(`SELECT id, name FROM languages`);
    let language_result = await databaseQueryModel.executeQuery("Games Model | GetAdminGames.fetchLanguage", fetchLanguageQuery);
    return {status: true, result: game_result, language_result: language_result};
  } catch(err) {
    return {status: false, message: "Failed to fetch admin games.", err: err};
  }
}

/* Function will fetch all games */
games.fetchAdminGames = async (req=undefined) => {
  return await GetAdminGames(undefined, req);
}

```

Note how much better it is to read codes that are written using async/await?

Callback Exercises

As a way for you to get a bit more comfortable with callbacks, let's have you work on the following exercises.

1. Create a function that takes another function as its argument. Have the function execute the passed function.
2. Create a function that returns a function. Have the returned function be executed.
3. Create a function that takes two functions as its arguments. Randomly, either execute the first function or the second function.

Once you've completed the assignments above, then you're ready for the next step.

I have created additional challenges that require a knowledge of callback and put them in Hacker Hero's Advance Javascript Course: <https://www.hackerhero.com/advanced-javascript-beta>. Please start from the very first challenge and complete all challenges. The first five challenges will be a review of OOP concepts you've already learned. The next 11+ assignments will be focused on callbacks and other advanced features of Javascript.

Once you're done, take a screenshot of the map page (where it shows how many stars you've earned for each challenge). You will find at least 16 challenges in that Hacker Hero course.

Promise and Async/Await

As a way of practicing syntaxes for promise as well as async/await, let's have you build out a program that does the following:

Create a function called `EmitRandomNumber()`. In this function, after 2 full seconds (2000 ms), have it generate a random number between 0 to 100. If the random number generated is below 80, have it call that function again, up to 10 times, until the random number generated is greater than 80.

After the program is run, have it generate a log such as follows:

```
Attempt #1. EmitRandomNumber is called.  
2 seconds have lapsed.  
Random number generated is 35.  
- - - - -  
Attempt #2. EmitRandomNumber is called.  
2 seconds have lapsed.  
Random number generated is 76.  
- - - - -  
Attempt #3. EmitRandomNumber is called.  
2 seconds have lapsed.  
Random number generated is 53.  
- - - - -  
Attempt #4. EmitRandomNumber is called.  
2 seconds have lapsed.  
Random number generated is 85!!!  
- - - - -
```